Develop a Multicore Application for RV32M1-VEGA

1. Introduction

This document gives the basic information of how to develop a multicore application for RV32M1-VEGA device. The RV32M1 device integrates four cores: a RISC-V RI5CY core, a RISC-V ZERO-RISCY, an ARM Coretex-M4F, and an ARM Coretx-M0+. There's a JTAG port used to debug two RISC-V cores. To support multicore system, the modules including Messaging Unit (MU), Semaphores2 (SEMA42), Miscellaneous System Control Module (MSCM), and Extended Resource Domain Controller (XRDC) are integrated into the RV32M1 device.

The SDK of RV32M1-VEGA board provides not only the low-level driver APIs for multicore related modules but also the middleware libraries that provide the services and IPC protocol for multicore system. With the drivers and/or middleware libraries, developing a multicore system on RM32M1 is easy. This document isn't the documentation for the APIs of the drivers and the middleware libraries to support the multicore system.

This document summarizes the hardware features on the RV32M1 device to support multicore system, middleware libraries based on the MU including Multicore Manager (MCMGR) and PRMsg_lite, and demonstrates how to develop a multicore application using low-level driver APIs and/or middleware libraries.

Contents

2. Multicore Overview

2.1 Core modules

RV32M1 device has one ARM Cortex-M0+ core (CM0+), one ARM Cortex-M4F core (CM4), one RISC-V RI5CY core, and one RISC-V ZERO_RISCY core. Each core can run at 48 MHz in RUN mode or at 72 MHz in HSRUN mode.

RI5CY is a 4-stage in-order 32b RISC-V processor core. The ISA of RI5CY was extended to support multiple additional instructions including hardware loops, post-increment load and store instructions and additional ALU instructions that are not part of the standard RISC-V ISA.

ZERO-RISCY is a 2-stage in-order 32b RISC-V processor core. ZERO-RISCY has been designed to be small and efficient.

Four cores are divided into two separate core subsystems: CM4F/RI5CY and CM0+/ZERO-RISCY. The ARM CM4F and RI5CY are multiplexed to master ports of CM4F/RI5CY domain Crossbar Switch. The ARM CM0+ core and ZERO_RISCY core are multiplexed to master ports of CM0+/ZERO-RISCY domain Crossbar Switch;

2.2 System Flash and SRAM

The on-chip Flash includes two portions. One is with 1 MB size mainly for CM4F/RI5CY subsystem beginning at address 0x0000_0000. Another is with 256 KB size mainly for Cortex-M0+/ZERO-RISCY subsystem beginning at address 0x0100_0000. See the Figure 1.

All CM4F/RI5CY and CM0+/ ZERO-RISCY cores can access the entire amount of flash on the device with one exception – CM0+/ ZERO-RISCY cannot access lowest 4 KB region of CM4F/RI5CY flash starting from address 0x0000_0000. The cores in a subsystem can access to the flash associated with the other subsystem but it has lower performance.

There’re 5 blocks of SRAM on RV32M1 device. The first one is 64 KB mainly for CM4F/RI5CY subsystem ITCM SRAM beginning at address 0x0800_0000. The second block is 128 KB mainly for CM0+/ ZERO-RISCY subsystem TCM SRAM beginning at address 0x0900_0000. The third block has 192 KB manly for CM4F/RI5CY subsystem DTCM SRAM beginning at address 0x2000_0000. The last two are 4 KB and 2 KB SRAM elaborate for FlexRAM and USB use respectively. Each core in a subsystem accessing to the SRAM associated with another subsystem has low performance.

2.3 System Boot and Reset

RI5CY boots from the Flash address 0x000F_FF00. The last 256 bytes at the bottom of the 1M-Flash is used as RI5CY interrupt vector table (IVT) that includes interrupt ISRs and system exception handlers. The interrupt ISR table is located from 0x000F_FF00 to 0x000F_FF7F. The system exception handlers table is from -x000F_FF80 to 0x000F_0x000F_FFFF. The first exception handler is for Reset. So the initial PC after RI5CY reset is at 0x000F_FF80, as showed in the Figure 1.

ZERO-RISCY boots from the Flash address 0x0103_FF00. This is the last 256 bytes at the bottom of the 256K-Flash for the interrupt vector table including the interrupt ISRs and system exception handlers. The vector table is from 0x0103_FF00 to 0x0103_FFFF. Initial PC is at 0x0103_FF80.

CM4F boots from the Bootloader ROM address 0x0880_0000 or flash address 0x0000_0000. CM0+ boots from the BROM address 0x0880_0000 or flash address 0x0100_0000.

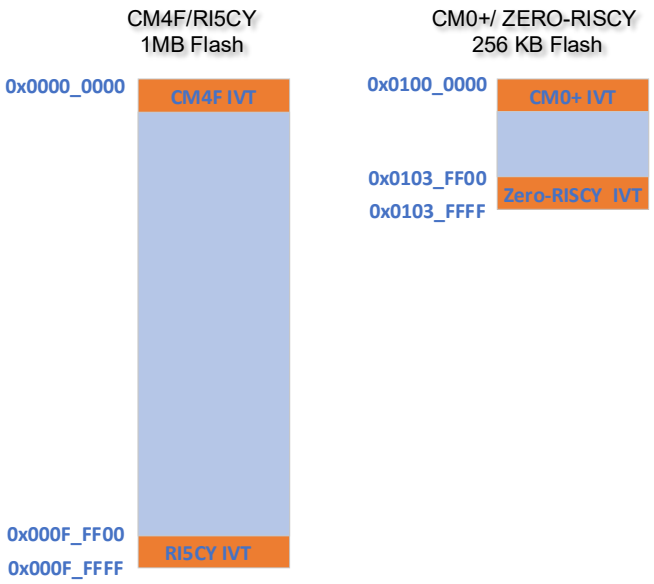

Figure 1. Flash address in RV32M1 map

The Flash Option (FOPT) in the flash IFR memory space with record index 0x84 defines the operation of the RV32M1 at boot time. The IFR bits, "CM4_EN, RI5_DIS, CM0P_EN, ZERO_DIS" in Table 1 control the boot up for CM4 or RI5CY, CM0+ or ZERO-RISCY. When a core is configured disabled in the FOPT, its clock will be gated off after reset.

0x84	FOPT	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48
		Reserved								Reserved							
		47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
		Reserved								Reserved							
		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
		~INSTANCEID[3:0]											TSTMR_EN				NMI_DIS
		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
		CM4_EN	RI5_DIS	CM0P_EN	ZERO_DIS	BOOTSRCSEL[1:0]		BOOTPINOPT			BOOTCORESEL		BOOTMODE				BOOTCLK

Table 1. Flash Option bit definitions

The Table 2 lists all the boot up modes supported by RV32M1 device. Considering the performance and the hardware limitation to support 3 or more cores run at the same time, only the highlighted modes are recommended but not limited to use.

{CM4_EN, RI5_DIS, CM0P_EN, ZERO_DIS}	BOOT CORE SEL	CM4 Clock Reset	RI5CY Clock Reset	CM0+ Clock Reset	Zero-riscy Clock Reset	Comments
1111	1	EN	DIS	EN	DIS	Boot CM4 firstly. MU enables CM0+ later. (Recommended)
1110	1	EN	DIS	EN	EN	Boot CM4 firstly. MU enables CM0+, Zero-riscy later.
1101	1	EN	DIS	DIS	DIS	Boot CM4 firstly. MU enables nothing.
1100	1	EN	DIS	DIS	EN	Boot CM4 firstly. MU enables Zero-riscy later.
1011	1	EN	EN	EN	DIS	Boot CM4, RI5CY firstly. MU enables CM0+ later.
1010	1	EN	EN	EN	EN	Boot CM4, RI5CY firstly. MU enables CM0+, Zero-riscy later.
1001	1	EN	EN	DIS	DIS	Boot CM4, RI5CY firstly. MU enables nothing.
1000	1	EN	EN	DIS	EN	Boot CM4, RI5CY firstly. MU enables Zero-riscy later.
01--	1	DIS	DIS	DIS	DIS	NA
0011	1	DIS	EN	EN	DIS	Boot RI5CY firstly. MU enables CM0+ later.
0010	1	DIS	EN	EN	EN	Boot RI5CY firstly. MU enables CM0+, Zero-riscy later.
0001	1	DIS	EN	DIS	DIS	Boot RI5CY firstly. MU enables nothing.
0000	1	DIS	EN	DIS	EN	Boot RI5CY firstly. MU enables Zero-riscy later. (Recommended)
1111	0	EN	DIS	EN	DIS	Boot CM0+ firstly. MU enables CM4 later. (Recommended)
1011	0	EN	EN	EN	DIS	Boot CM0+ firstly. MU enables CM4, RI5CY later.
0111	0	DIS	DIS	EN	DIS	Boot CM0+ firstly. MU enables nothing.
0011	0	DIS	EN	EN	DIS	Boot CM0+ firstly. MU enables RI5CY later.
1110	0	EN	DIS	EN	EN	Boot CM0+, Zero-riscy firstly. MU enables CM4 later.
1010	0	EN	EN	EN	EN	Boot CM0+, Zero-riscy firstly. MU enables CM4, RI5CY later.
0110	0	DIS	DIS	EN	EN	Boot CM0+, Zero-riscy firstly. MU enables nothing.
0010	0	DIS	EN	EN	EN	Boot CM0+, Zero-riscy firstly. MU enables RI5CY later.
--01	0	DIS	DIS	DIS	DIS	NA
1100	0	EN	DIS	DIS	EN	Boot Zeor-riscy firstly. MU enables CM4 later.
1000	0	EN	EN	DIS	EN	Boot Zeor-riscy firstly. MU enables CM4, RI5CY later.
0100	0	DIS	DIS	DIS	EN	Boot Zeor-riscy firstly. MU enables nothing.
0000	0	DIS	EN	DIS	EN	Boot Zeor-riscy firstly. MU enables RI5CY later. (Recommended)

Table 2. RV32M1 boot up modes

3. Multicore Unit on RV32M1

Multicore Unit on RV32M1 includes Messaging Unit (MU), Semaphores2 (SEMA42), and Extended Resource Domain Controller (XRDC) that provide hardware support for multicore system with communication and services. This section gives an overview of their features

3.1 Messaging Unit

The Messaging Unit (MU) module enables two processors within the SoC to communicate and coordinate by passing messages (e.g. data, status and control) through the MU interface. The MU also provides the ability for one processor to signal the other processor using interrupts.

The MU module contains two sets of registers (Processor A-facing, Processor B-facing). The Processor A in the RV32M1 refers to the CM4F or RI5CY and the Processor B refers to CM0+ or ZERO-RISCY. They work in synchronized clock domains. The block diagram is showed in the Figure 2.

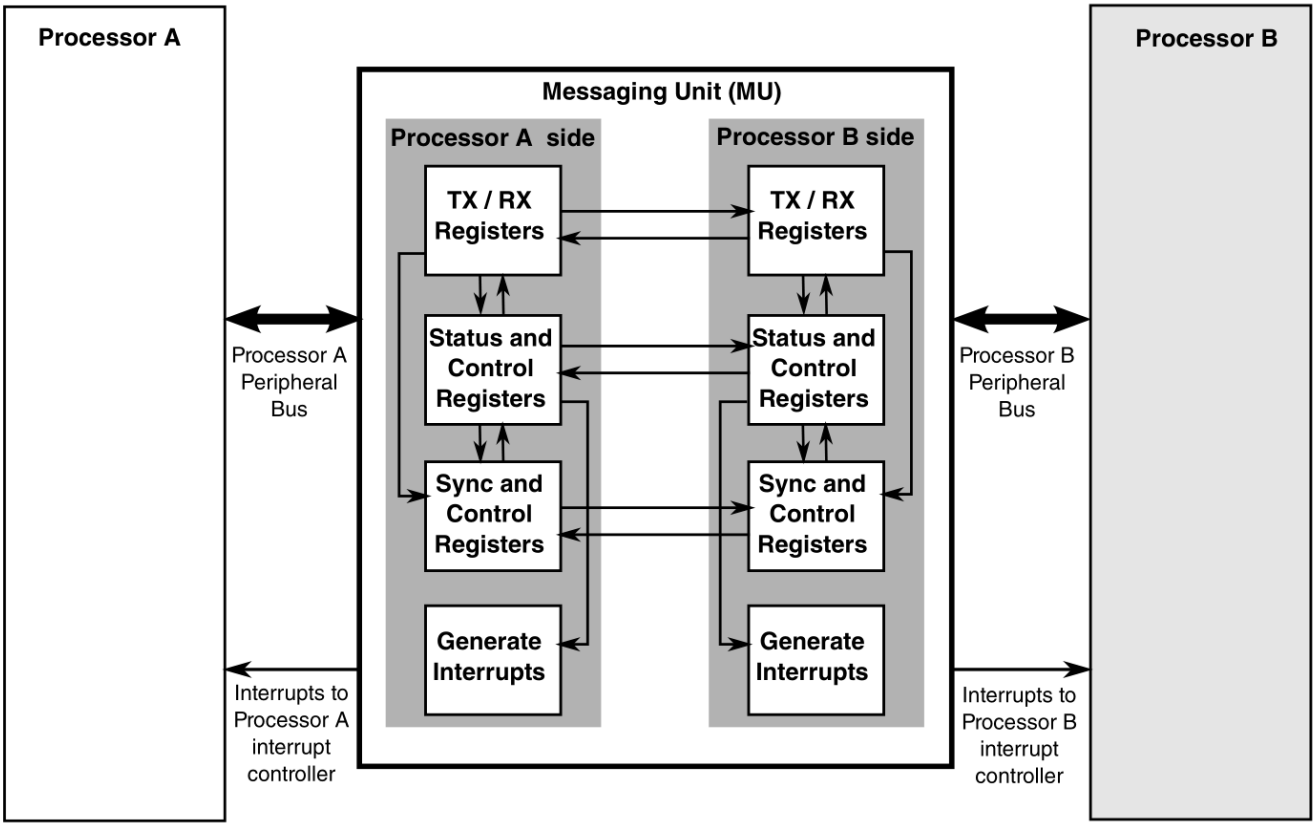

Figure 2. MU Block Diagram

3.1.1 MU Messaging

The MU provides 32-bit status and control registers to the Processor B and Processor A sides for control operations (such as interrupts and reset), and for status checking of the other MU-side.

For messaging, the MU has four, 32-bit write-only transmit registers and four, 32-bit read-only receive registers on the Processor B and Processor A-sides. These registers are used for sending messages to each other. These messages can also be controlled using the 3 general purpose flags provided in the control and status registers of either MU-side.

3.1.2 Programmer Model

The messaging logic is used in conjunction with external memory. You have various messaging methods, which you can use to implement a messaging protocol. Some of these messages could mean “I have just written a message of N words, starting at offset X in the memory,” or “I have just finished reading the previous data block that was sent.” Having the messaging logic independent from the memory array does not restrict you to a predefined hardware protocol. On the other hand, the software needed to manage the messaging is short and straightforward.

Most of the messaging mechanisms are symmetric; they are duplicated and are available on both the Processor B-side and the Processor A-side. The messaging mechanisms are:

- Four, 32-bit write-only transmit registers, which are each reflected in four, read-only receive registers in the other processor's side. You can use these registers to transfer 32-bit word messages or frame information of messages written to the shared memory (number of words, initial address, and message type code).
- A write to a transmit register on the transmitter side clears a "transmitter empty" bit in the Status Register on the transmitter side, and sets a "receiver full" bit in the Status Register on the receiver side. The setting of the bit at the receiver side can optionally trigger an interrupt at the receiver side (maskable receive interrupt).
- A read of one of the receive registers at the receiver side clears the "receiver full" bit in the Status Register at the receiver side, and sets the "transmitter empty" bit in the Status Register on the transmitter side. The setting of the "transmitter empty" bit can optionally trigger an interrupt at the transmitter side (maskable transmit interrupt).
- Four general purpose flags are reflected in the Status Register on the receiver side
- A read/write access to any reserved location and a write to a read-only register on the Processor A-side of the MU will generate a module transfer error acknowledge to the Processor A.
- A read/write access to any reserved location and write to a read-only register on the Processor B-side of the MU will generate a module transfer error acknowledge to the Processor B.

3.1.3 Messaging Examples

The following are messaging examples:

- **Passing short messages**: Transmit register(s) can be used to pass short messages from one to four words in length. For example, when a four-word message is desired, only one of the registers needs to have its corresponding interrupt enable bit set at the receiver side; the message's first three words are written to the registers whose interrupt is masked, and the fourth word is written to the other register (which triggers an interrupt at the receiver side).
- **Passing frame information:** Transmit registers can be used to pass frame information for long messages written to the shared system (SDRAM and SyncFLASH). Such frame information normally includes a start address, number of words, and perhaps a message type code.
- **Passing event notices and requests**: Events and requests that do not include data words can be signaled from the Processor B to the Processor A using the general interrupts, such as acknowledging that a long message was read from the shared system memory.
- **Passing fixed length data**: Formatted data with a fixed length can be written in predetermined locations in the shared memory. A processor can use a general interrupt (Processor A or Processor B) to signal the other processor that the data is ready.
- **Passing announcements:** The three flags can be used by a processor to announce its current program state or other billboard messages to the other processor.

3.1.4 MU Driver API

The RV32M1 SDK provides a driver for the MU module to facilitate the access to the MU registers and the development of dual cores application by providing the following functions:

- Functions to initialize the MU module.
- Functions to send and receive messages between two cores.
- Functions for MU flags for both MU sides.
- Functions for status flags and interrupts.
- Other miscellaneous functions.

The drive APIs just implements the communication between dual cores and the basic hardware operations to start/stop/reset another core and gate the clock of another core in the dual core system. For more information, see SDK API reference manual.

3.2 Semaphores2

The Semaphore2 (SEMA42) provides robust hardware support needed in multi-core systems for implementing semaphores and provides a simple mechanism to achieve "lock and unlock" operations via a single write access. The hardware semaphore module provides hardware-enforced gates as well as other useful system functions related to the gating mechanisms.

These gating mechanisms are used by the software to serialize (and synchronize) accesses to shared data and/or hardware resources to prevent race conditions and preserve memory coherency between different processes and processors.

3.2.1 Semaphores2 features

The RV32M1 cores can be assigned as different domain ID that’s used by Semaphore2 to identify the core. The Semaphore2’s gates appear as an 16-entry byte-size array with read and write accesses.

- Cores lock gates by writing "domain_id+1" to the appropriate gate and must read back the gate value to verify the lock operation was successful.
- Once locked, the gate can and must be unlocked by a write of zeroes from the locking processor.
- Multiple gate values can be read in a single access, but only a single gate can be updated via a write operation at a time.
- Semaphores module implements a "secure" reset mechanism that allows a hardware gate (or all the gates) to be initialized by following a specific dual-write access pattern that requires two consecutive writes with predefined data patterns from the same domain to force the clearing of the specified gate(s).

For more detailed feature of Semaphores2, refer to the RV32M1 reference manual.

3.2.2 Semaphore2 Driver API

The RV32M1 SDK provides a driver for the Semaphore2 module to safely and easily provide a locking mechanism that is used by system software to control access to shared data structures, shared hardware resources, and so on.

The API includes functions to initialize and deinitialize the module, reset gate(s), get the gate status, and lock/unlock the gates. The driver provides two functions to lock the SEMA42 gate. One lock function tries to lock the gate. This function returns an error immediately if the gate has been locked by another processor. Another lock function is a blocking method, which waits until the gate is free and locks it.

3.3 Extended Resource Domain Controller

The Extended Resource Domain Controller (XRDC) mainly is for security and provides an integrated, scalable architectural framework for access control, system memory protection and peripheral isolation. It allows software to assign chip resources including processor cores, non-core bus masters, memory regions and slave peripherals to processing domains to support enforcement of robust operational environments.

First, each bus mastering resource is assigned to a domain identifier (domainID, DID). For processors, there are additional fields that can optionally be used to assign it to multiple domains. Next, the access control policies for the individual domains are programmed into any number of slave memory region descriptors and slave peripheral domain access control registers. Finally, all accesses throughout the device are monitored concurrently to determine the validity of each access. If a reference from a given domain has sufficient access rights, it is allowed to continue, else the access is aborted, and error information captured.

3.3.1 XRDC in Multicore System

In previous subsection, the value of domain ID+1 is used to write Semaphore2's gate register to lock a gate. To use Semaphores2 for semaphore lock/unlock, XRDC is needed to work with Semaphores2. The features of XRDC used for multicore are the master domain assignment, access right to slave targets, and access controls for peripherals. RV32M1 XRDC configuration supports up to 3 domain ID. Now SDK only supports dual core system even there're four cores and three domain ID.

The typical use case related to maser domain assignments in a multicore system is to include each processor core in a single domain, configure a domain ID to access peripherals with specific policy, for example, to configure the 1 MB flash only accessible for the CM4F or RI5CY core and inaccessible for the CM0+ or ZERO-RISCY core.

3.3.2 XRDC Driver API

The XRDC module includes four submodules, as follows:

- XRDC_MGR. The Manager submodule coordinates all programming model reads and writes.
- XRDC_MDAC The Master Domain Assignment Controller handles resource assignments and generation of the domain identifiers (domain ID).

- XRDC_MRC The Memory Region Controller implements the access controls for slave memories based on the pre-programmed region descriptor registers.
- XRDC_PAC The Peripheral Access Controller implements the access controls for slave peripherals based on the preprogrammed domain access control registers.

Accordingly, the XRDC driver functions could be grouped as follows:

- XRDC_MGR functions.
- XRDC_MDAC functions.
- XRDC_MRC functions.
- XRDC_PAC functions.

4. Multicore Middleware Libraries

Besides the driver APIs for multicore related modules, RV32M1 SDK also supports two middleware libraries including Multicore Manager(MCMGR) and RPMsg-Lite to provide the services for multicore system and communication between multi cores in the multicore system.

4.1 Multicore Manager

The Multicore Manager (MCMGR) software library provides a few services for multicore systems. The services provided by MCMGR uses the MU functions and its implementation is based on the MU driver API.

Since the MU is designed for the dual core system. The Multicore Manager can be only used for the dual core application even there're four cores in the RV32M1 device. The Core0 of MU refers to CM4F or RI5CY and the Core1 refers to CM0+ or ZERO_RISCY. Using MU for multicore system, only one core can be enabled in each CM4F/RI5CY and CM0+/ZERO-RISCY subsystem.

The Multicore Manager provides the following three types of functions:

1. Maintains information and status about all cores in the system.

   These APIs provides the functions to initialize MCMGR, get the information of core including the count of cores in a multicore system, current core id, property of a core.

2. Secondary/auxiliary core startup and shutdown.

   There're two API: MCMGR_StartCore() causes a selected core to initialize and start the code execution, MCMGR_StopCore causes a selected core to halt the code execution.

3. Remote core monitoring and even handling.

   MCMGR uses the event mechanism for the communication between dual cores and the services in the multicore system. A core registers an even with a callback function. Another core will trigger that event to notice the first core. The first core will respond to the triggered event by calling the registered callback function. Only kMCMGR_RemoteApplicationEvent event is defined for the application use. All other predefined events are used by MCMGR and RPMsg_Lite. But the total event number can be extended by adding more enum values in the mcmgr_event_type_t definition.

4.2 Lightweight Implementation of RPMsg

The RPMsg_Lige is a lightweight implementation of the Remote Processor Messaging (RPMsg) protocol that defines a standardized binary interface used to communicate between multiple cores in a heterogeneous multicore system. Compared with the standard RPMsg implementation, the RPMsg_Light has a reduced code size, simplified API, and improved modularity.

The RPMsg_Light is open-source including three sub-components, from which two are optional. The core component is located in rpmsg_lite.c. Two optional components are used to implement a blocking receive API (in rpmsg_queue.c) and dynamic "named" endpoint creation and deletion announcement service (in rpmsg_ns.c).

The actual "media access" layer is implemented in virtqueue.c, which is one of the few files shared with the OpenAMP implementation. This layer mainly defines the shared memory model, and internally defines used components such as vring or virtqueue.

The porting layer is split into two sub-layers: the environment layer and the platform layer. The first sublayer is to be implemented separately for each environment. (The bare metal environment already exists and is implemented in rpmsg_env_bm.c, and the FreeRTOS environment is implemented in rpmsg_env_freertos.c etc.) Only the source file, which matches the used environment, is included in the target application project. The second sublayer is implemented in rpmsg_platform.c and defines low-level functions for interrupt enabling, disabling, and triggering mainly. The situation is described in the Figure 3:

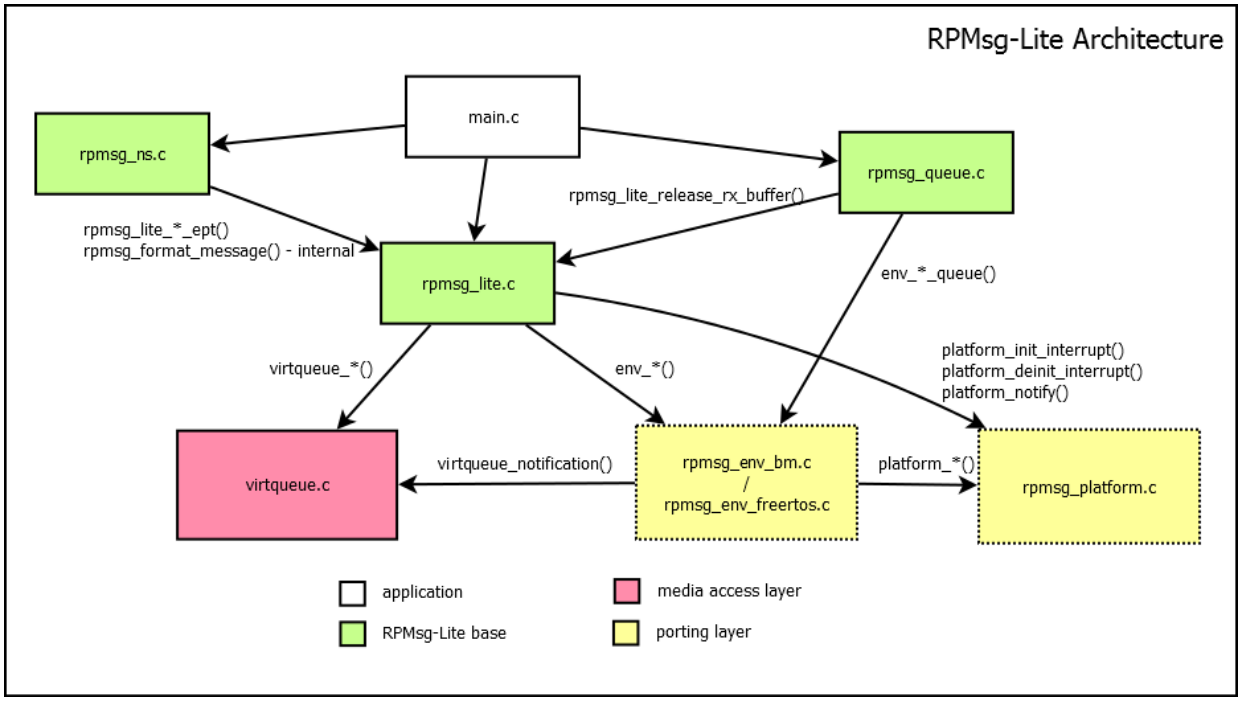

Figure 3. RPMsg-Lite Architecture

1. RPMsg-Lite core sub-component

   This subcomponent implements a blocking send API and callback-based receive API. The RPMsg protocol is part of the transport layer. This is realized by using so-called endpoints. Each endpoint can be assigned a different receive callback function. However, it is important to notice that the callback is executed in an interrupt environment in current design. Therefore, certain actions like memory allocation are discouraged to execute in the callback. The following figure shows the role of RPMsg in an ISO/OSI-like layered model as in the Figure 4::

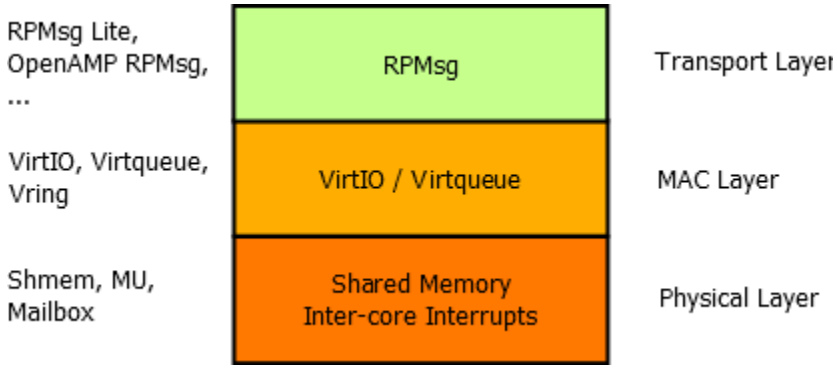

Figure 4. RPMsg ISO/OSI Layered Model

2. Queue sub-component (optional)

   This subcomponent is optional and requires implementation of the env_*_queue() functions in the environment porting layer. It uses a blocking receive API, which is common in RTOS-environments. It supports both copy and nocopy blocking receive functions.

3. Name Service sub-component (optional)

   This subcomponent is a minimum implementation of the name service which is present in the Linux Kernel implementation of RPMsg. It allows the communicating node both to send announcements about "named" endpoint (in other words, channel) creation or deletion and to receive these announcements taking any user-defined action in an application callback. The endpoint address used to receive name service announcements is arbitrarily fixed to be 53 (0x35).

5. Multicore Application using Native driver API

This section tells how to develop a multicore application using the native driver APIs. According to the description in the above sections, the MU module itself can alone support multicore system. But SEMA42 only is the hardware to be used to implement an exclusive access to the share data and/or hardware resources. Before using SEMA42 to lock the gate, the core shall be assigned a domain ID through XRDC, then writes the domain ID plus 1 to the gate to request the lock.

5.1 Multicore application using MU

The Flash Option (FOPT) defines system boot up mode. There’re 4 cores in the RV32M1 device, but each MU side only supports one core. So, MU only supports dual core application. The first enabled core refers to the primary core (core0) and another is the secondary core (core1) in this document.

Application code run on the core0 should initialize the MU by calling MU_Init(), then boots the core1 with a boot configuration by calling MU_BootOtherCore() or MU_BootCoreB(). Now core0 waits the notice from core1 to indicate the core1's bootup done.

The code on the core1 should initialize the MU first then notices core0 that core1 is boot up. After the core0 gets the notice of core1 bootup done, the core0 can communicate with core1. As showed in the Figure 5:

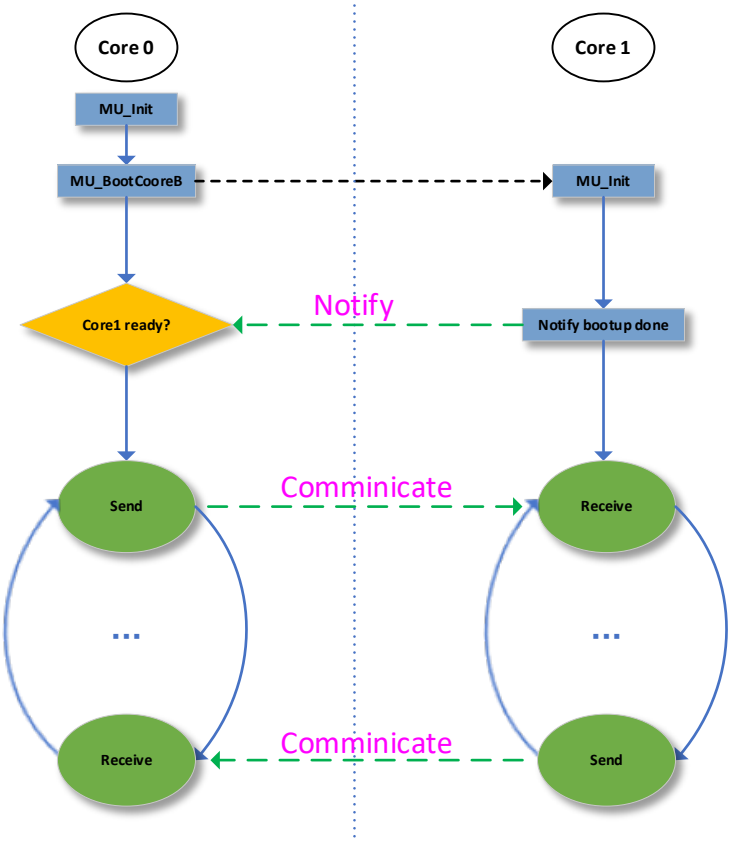

Figure 5. Multicore scenarios using MU

There're many ways to implement the communication between two cores. Some messaging examples are given in section 3.1.3. User can use driver APIs to implement one of these messaging examples. The simplest communication is to set the other MU side flag. For example, the core1's bootup done can be noticed to core0 by setting the specific flag of MU side corresponding to the core0 in the core1's code. The core0's code checks its flag till the flag is set by core1 to make sure the bootup of core1 is finished. The MU_SetFlags() and MU_GetFlags() are for such purpose.

The second communication is to send/receive message using TX/RX register in the MU. This communication can be done in the polling mode or interrupt mode. The polling mode is to write/read the TX/RX register directly and the message isn't sent until the TX register is empty and the message isn't received until the RX is full. The interrupt mode is that the message is sent or received in the interrupt service routine of MU. When there's a writing/reading TX/RX register in a MU side then there'll be an interrupt generated in the other MU side whose ISR needs to read the RX register to get the message from the first side or write the TX register to send the next message to the first side. The default MU ISR

is implemented as a weak function. The strong version for MU ISR is needed to implement the interrupt mode communication.

The reference code of both polling mode and interrupt mode can be found under <sdk_install>\boards\rv32m1_vega\driver_examples\mu.

5.2 Multicore application using SEMA42

The multicore system may have the shared memory or hardware resources. It's convenient to use the Semaphores module to lock the shared object before accessing it and unlock it to release it for the other core access.

The SEMA42 API only provides the lock and unlock function. It needs the MU and XRDC to work together to support a multicore system. The application should initialize the MU module that'll be used to boot up the second core.

The application run on the core0 should have the following steps:

- Call the MU_Init() to initialize the MU
- Call XRDC_Init() to initialize the XRDC
- Assign the processor domain ID by calling XRDC_SetProcessorDomainAssignment()
- Configure the peripheral access level if needed
- Call XRDC_SetGlobalValid() with true to set the XRDC global valid.
- Call SEMA42_Init() to initialize the Sema42
- Call SEMA42_ResetAllGates() to reset the Sema42 gates
- Call MU_BootCoreB() to boot up the core1
- Wait for the notice that the core1 is boot up
- Call SEMA42_Lock() before accessing the shared hardware resources
- Call SEMA42_Unlock() after finishing the access to the shared object

The code run on the core1 is relative simple compared with the code on core0 since XRDC related management is done in the code of core0. The following is the steps of core1:

- Call the MU_Init() to initialize the MU
- Notify the core0 that bootup is finished
- Call SEMA42_Lock() before accessing the shared hardware resources
- Call SEMA42_Unlock() after finishing the access to the shared object

The user is responsible for guaranteeing the XRDC initialization and domain id assignment done before SEMA42 initialization. The Figure 6 shows the steps given above.

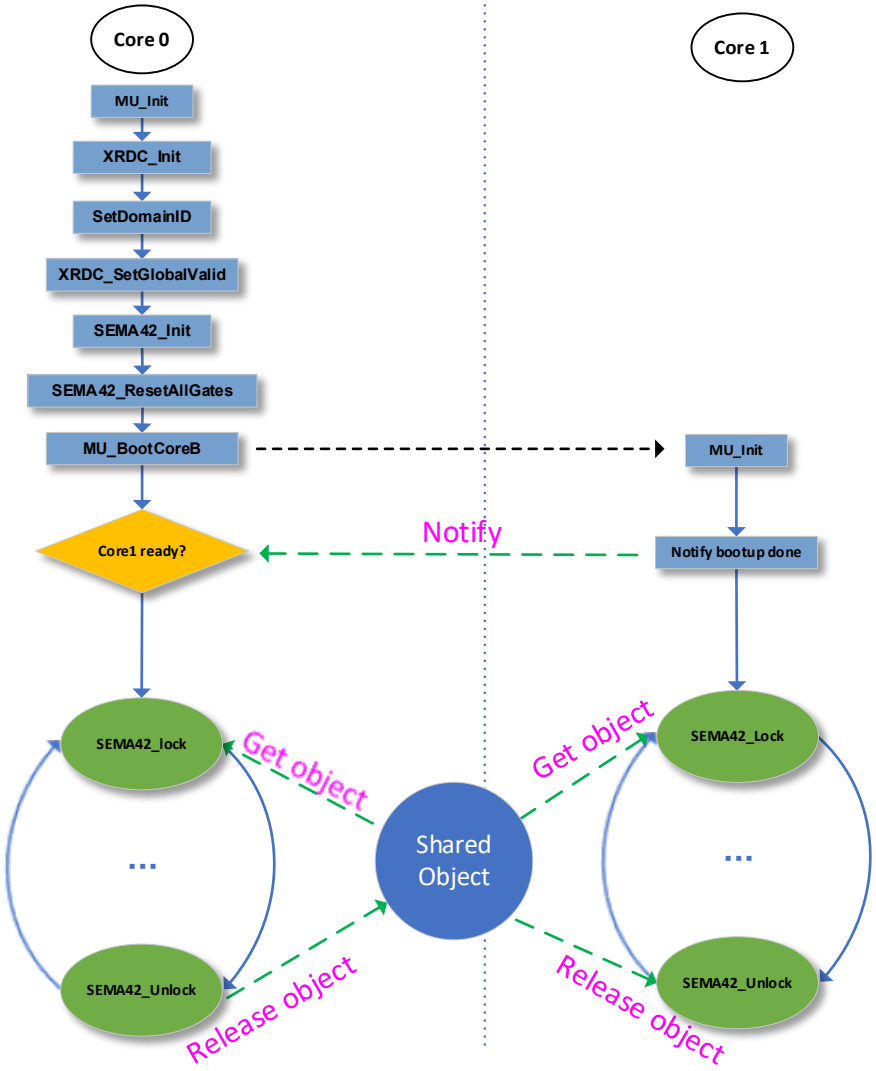

Figure 6. Multicore scenarios using SEMA42

The reference code using SEMA42 to implement multicore application can be found in the <sdk_install>\boards\rv32m1_vega\driver_examples\sema42\dual_core folder.

6. Multicore Application using Middleware API

Using native driver APIs to develop the multicore system is still difficult because user is responsible for maintaining the high-level protocol of communication between multiple processors. For example, calling MU_SendMsg() does not check whether the RX register is full or not. The application code should make sure the RX register is full before calling this function.

RV32M1 SDK provides two middleware libraries the Multicore Manager and RPMsg_Lite that facilitate the multicore application development. The following subsections will describe how to use them to develop a multicore system.

6.1 Multicore application using MCMGR

The Multicore Manager (MCMGR) is based on the MU APIs and provides an event-based communication. Compared with the native MU APIs, MCMGR APIs are straightforward and easy to use. User can them without the knowledge of MU hardware.

Application code run on the primary core should use the following steps：

- Call MCMGR_EarlyInit() to initialize MCMGR. This function should be call as close to the reset entry as possible to allow CoreUp Event triggering.
- Call MCMGR_Init() to install generic event handlers.
- Register the handlers of the events that the current core will process. The event will be triggered by the code of the other core. MCMGR_RegisterEvent() can be called one or more times that depends on the number of events processed by the current core.
- Register the handler of the kMCMGR_RemoteApplicationEvent event that is defined for the application use only. User can use it for different application specific purposes distinguished by the event data, for example, for notifying the primary core that the secondary core is boot up with a special event data value. Its handler must save the event data from the secondary core for primary core to check.
- Call MCMGR_StartCore() to boot the secondary core application.
- Check the event data saved in the kMCMGR_RemoteApplicationEvent handler to make sure the secondary core is boot up.
- Use a switch-case or if-else statement to process the event data saved by the event handler.
- Call MCMGR_TriggerEvent() to communicate with the secondary core by triggering the event handler registered by the code of the secondary core.

Application code run on the secondary core should have the steps like to the following:

- Call MCMGR_EarlyInit() to initialize MCMGR. This function should be call as close to the reset entry as possible to allow CoreUp Event triggering.
- Call MCMGR_Init() to install generic event handlers.
- Call MCMGR_RegisterEvent() to register the handler of the event that the current core will process.
- Call MCMGR_GetStartupData() until returning the success value. This call initiates the whole communication between two cores.
- Call MCMGR_TriggerEvent() to trigger the event registered by the primary core to have event handler called in the primary core.
- Use a switch-case or if-else statement to process the event data saved by the vent handler.

The following Figure 7 shows the steps described above:

Figure 7. Multicore scenarios using MCMGR

The reference code using MCMGR to implement multicore application can be found in the <sdk_install>\boards\rv32m1_vega\multicore_examples\multicore_manager.

6.2 Multicore application using RPMsg_Lite

It can be observed from the scenarios of multicore system introduced above that the MU module is always needed in the multicore system. Because the MU provides the core0 a way to boot up the core1. User can select to use the MU driver API or MCMGR middleware in his application to boot up the other core. Using MU to boot up the other core has the same code scenarios as that in previous examples. This section will not cover them anymore, instead, focus on using RPMsg_lite middleware to implement the communication between two cores.

The first step is to initialize the stack by calling the rpmsg_lite_master_init() on the master side (core0) and the rpmsg_lite_remote_init() on the remote side (core1). This initialization function must be called prior to any RPMsg-Lite API call. After the init, it is wise to create a communication endpoint, otherwise communication is not possible. This can be done by calling the rpmsg_lite_create_ept() function. It optionally accepts a last argument, where an internal context of the endpoint is created, just in case the RL_USE_STATIC_API option is set to 1. If not, the stack internally calls env_alloc() to allocate dynamic memory for it. In case a callback-based receiving is to be used, an ISR-callback is registered to each new endpoint with user-defined callback data pointer. If a blocking receive is desired (in case of RTOS environment), the rpmsg_queue_create() function must be called before calling rpmsg_lite_create_ept(). The queue handle is passed to the endpoint creation function as a callback data argument and the callback function is set to rpmsg_queue_rx_cb(). Then, it is possible to use rpmsg_queue_receive() function to listen on a queue object for incoming messages. The rpmsg_lite_send() function is used to send messages to the other side.

The RPMsg-Lite also implements no-copy mechanisms for both sending and receiving operations. These methods require specifics that have to be considered when used in an application.

no-copy-send mechanism: This mechanism allows sending messages without the cost for copying data from the application buffer to the RPMsg/virtio buffer in the shared memory. The sequence of no-copy sending steps to be performed is as follows:

- Call the rpmsg_lite_alloc_tx_buffer() function to get the virtio buffer and provide the buffer pointer to the application.
- Fill the data to be sent into the pre-allocated virtio buffer. Ensure that the filled data does not exceed the buffer size (provided as the rpmsg_lite_alloc_tx_buffer() *size* output parameter).
- Call the rpmsg_lite_send_nocopy() function to send the message to the destination endpoint. Consider the cache functionality and the virtio buffer alignment. See the rpmsg_lite_send_nocopy() function description below.

no-copy-receive mechanism: This mechanism allows reading messages without the cost for copying data from the virtio buffer in the shared memory to the application buffer. The sequence of no-copy receiving steps to be performed is as follows:

- Call the rpmsg_queue_recv_nocopy() function to get the virtio buffer pointer to the received data.
- Read received data directly from the shared memory.
- Call the rpmsg_queue_nocopy_free() function to release the virtio buffer and to make it available for the next data transfer.

The user is responsible for destroying any RPMsg-Lite objects he has created in case of deinitialization. In order to do this, the function rpmsg_queue_destroy() is used to destroy a queue, rpmsg_lite_destroy_ept() is used to destroy an endpoint and finally, rpmsg_lite_deinit() is used to deinitialize the RPMsg-Lite intercore communication stack. Deinitialize all endpoints using a queue before deinitializing the queue. Otherwise, you are actively invalidating the used queue handle, which is not allowed. RPMsg-Lite does not check this internally, since its main aim is to be lightweight.

The following Figure 8 shows the steps described above:

Figure 8. Copy and no-copy interface, multicore scenarios using RPMsg_Lite

The reference code using RPMsg_Lite to implement multicore application can be found in the <sdk_install>\boards\rv32m1_vega\multicore_examples\rpmsg_lite_pingpong. The implementation using the blocking receive API can be found in the <sdk_install>\boards\rv32m1_vega\multicore_examples\rpmsg_lite_pingpong_rtos.

7 References

Following references are available on www.open-isa.org:

- *RV32M1RM: Reference Manual*
- *RV32M1_Vega_Develop_Environment_Setup*

8 Revision history

Rev.	Date	Substantive change(s)
0	10/2018	Initial release

.